Q1 Modified Dining Philosophers Problem

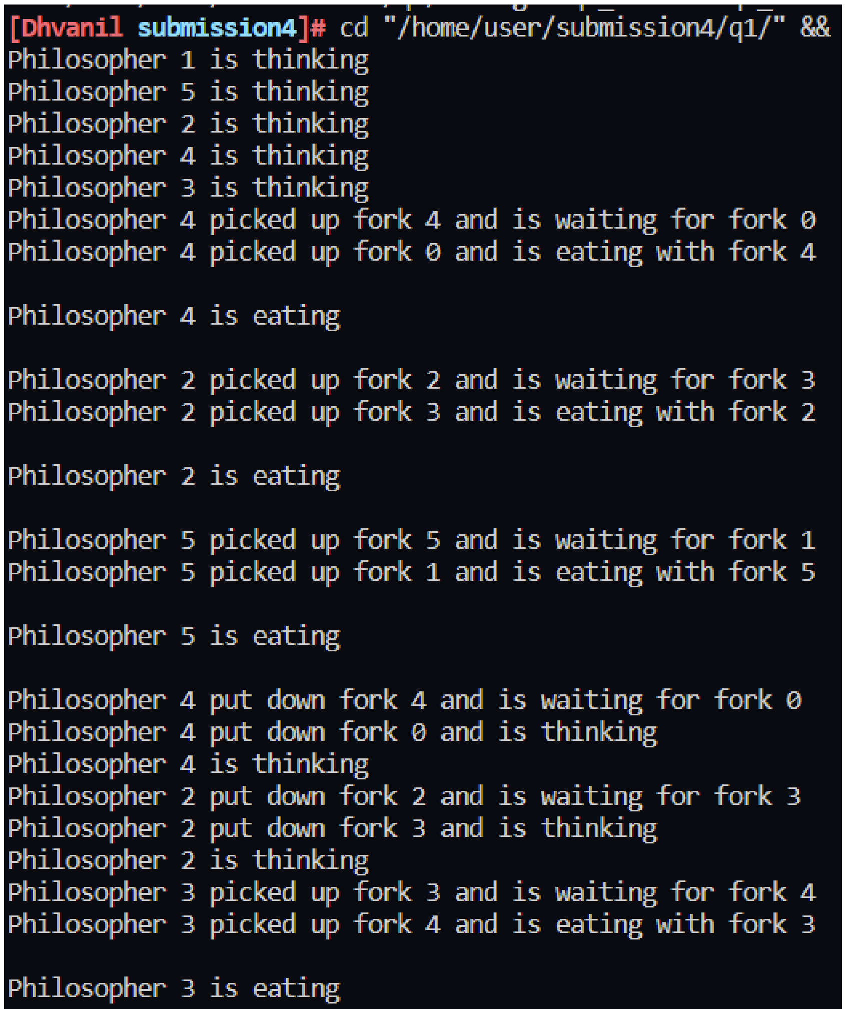
Q1 A)

This code is a solution to the Dining Philosophers problem.

It defines a function '**phil'** that will be run by each philosopher as a separate thread. This function takes a single argument, which is a pointer to an integer representing the philosopher's number. The function begins by declaring variables for the left and right forks that the philosopher will need to pick up in order to eat. It then enters an infinite loop in which the philosopher thinks for a period of time, and then attempts to pick up the left and right forks. If either of the forks is not available (indicated by a value of 1 in the **forks** array), the philosopher waits for a short time and then checks again.

The main function creates the philosopher threads and waits for them to finish. The availability of each fork is tracked using an array, with a value of 1 indicating that a fork is not available and a value of 0 indicating that it is.

Sample Output



```
[Dhvanil submission4]# cd "/home/user/submission4/q1/" &&
Philosopher 1 is thinking
Philosopher 5 is thinking
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 3 is thinking
Philosopher 4 picked up fork 4 and is waiting for fork 0
Philosopher 4 picked up fork 0 and is eating with fork 4

Philosopher 4 is eating

Philosopher 2 picked up fork 2 and is waiting for fork 3
Philosopher 2 picked up fork 3 and is eating with fork 2

Philosopher 2 is eating

Philosopher 5 picked up fork 5 and is waiting for fork 1
Philosopher 5 picked up fork 1 and is eating with fork 5

Philosopher 5 is eating

Philosopher 4 put down fork 4 and is waiting for fork 0
Philosopher 4 put down fork 0 and is thinking
Philosopher 4 is thinking
Philosopher 2 put down fork 2 and is waiting for fork 3
Philosopher 2 put down fork 3 and is thinking
Philosopher 2 is thinking
Philosopher 3 picked up fork 3 and is waiting for fork 4
Philosopher 3 picked up fork 4 and is eating with fork 3

Philosopher 3 is eating
```

Q1 B)

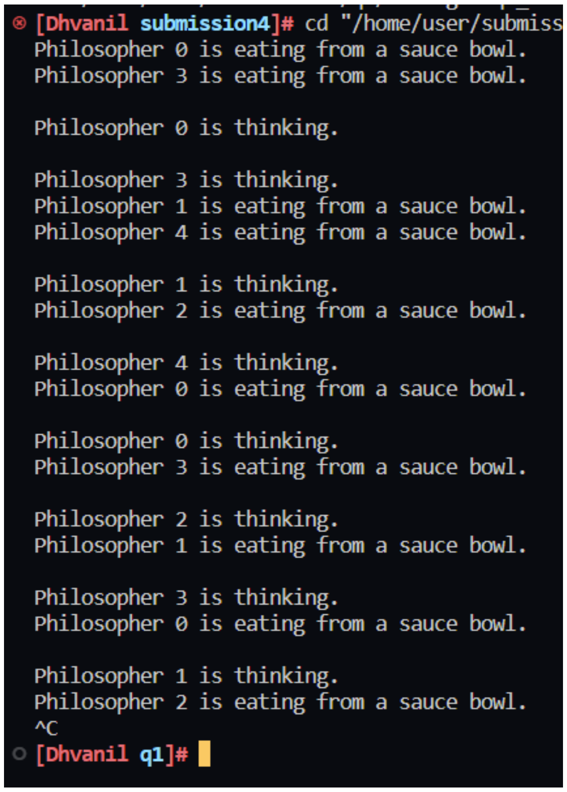
Philosophers and initializes the semaphores. It then creates a thread for each philosopher and waits for all threads to complete before destroying the semaphores and exiting.

To avoid deadlocks and ensure that all philosophers are able to eat eventually, the code uses a resource hierarchy and semaphores (for A &B). Specifically, each philosopher acquires their left fork before their right fork if they have an even ID, and vice versa if they have an odd ID. This ensures that no two philosophers will try to acquire the forks in the same order, which helps prevent deadlocks.

Additionally, the code uses the semaphore '**sauce_bowls'** to ensure that a maximum of two philosophers can eat from the sauce bowls at the same time. This helps prevent starvation, as it ensures that each philosopher has a chance to acquire the resources they need to eat.

Overall, it is a solution to a modified version of the dining philosopher's problem that involves two common sauce bowls and chopsticks. It uses semaphores and a resource hierarchy solution to avoid deadlocks and ensure that all philosophers are able to eat eventually.

Sample Output



```
⊗ [Dhvanil submission4]# cd "/home/user/submiss
  Philosopher 0 is eating from a sauce bowl.
  Philosopher 3 is eating from a sauce bowl.

  Philosopher 0 is thinking.

  Philosopher 3 is thinking.
  Philosopher 1 is eating from a sauce bowl.
  Philosopher 4 is eating from a sauce bowl.

  Philosopher 1 is thinking.
  Philosopher 2 is eating from a sauce bowl.

  Philosopher 4 is thinking.
  Philosopher 0 is eating from a sauce bowl.

  Philosopher 0 is thinking.
  Philosopher 3 is eating from a sauce bowl.

  Philosopher 2 is thinking.
  Philosopher 1 is eating from a sauce bowl.

  Philosopher 3 is thinking.
  Philosopher 0 is eating from a sauce bowl.

  Philosopher 1 is thinking.
  Philosopher 2 is eating from a sauce bowl.
  ^C
○ [Dhvanil q1]#
```